Tokenization Manager

For a list of all available token processors, see Token Processors.

For information on programming new token processors, see Creating Token Processors.

Also see the Use Cases below.

How It's Put Together

The basic operation of the Tokenization Manager component is shown in the diagram:

blocked URL

- The TokenizationManager component produces AspireAnalyzer objects.
- AspireAnalyzer objects take in string data, tokenize the data, and can then do further processing on the tokens.

There are two types of processing:

  - *Token manipulation* - filtering, modifying, or splitting tokens
  - *Computing summary information* - for example, counting tokens, creating token histograms

Inputs to the token processing pipeline

Text can be sent into the tokenizer in two ways:

1. A groovy stage can call processAll() (or a similar process method) on the AspireAnalyzer to process specific text.
2. The TokenizationManager can be a pipeline stage, in which selected tags from the job's XML will be automatically sent into the analyzer

Outputs from the token processing pipeline

Output can be retrieved from the tokenization pipeline in two ways

1. A groovy stage can get a token stream, and can then extract tokens from the pipeline one at a time
   - **(some work on this is still TBD)**
2. The results of some token processors - specifically those that create summary statistics, can be automatically stored in variables, either a job's variables or a job's parent's variables (see below)

Aspire Analyzers Are Different than Lucene Analyzers

Note that an "AspireAnalyzer" works differently than analyzers in Lucene. In Lucene, analyzers are factories for TokenStreams, from which tokens can be retrieved.

In Aspire, the TokenizationManager manufactures AspireAnalyzer objects. Each AspireAnalyzer contains a Lucene Token Stream, i.e. an instance of a token pipeline. Each AspireAnalyzer object can only be used by a single thread to process text data into tokens.

Summary Statistics and Scope

Many token processing modules produce summary statistics on tokens. Different types of summary statistics include:

- Counting tokens
- Histogram of token occurrences - number of occurrences for every unique token in the document
- Token document histogram - for every unique token, the number of documents which contain it
- Token statistics - a complete summary of statistics for every unique token
- Document score - computes a score for the document based on the scores of the tokens it contains

All of these summary statistics depend on a "scope" - which determines over what set of strings, fields, or documents the statistics are computed. The different scopes are as follows:

- **Analyzer scope** - Accumulates statistics across all text processed by the same AspireAnalyzer object.
- **Document Scope** - Accumulates statistics across the document (or sub-job). In addition:
  1. Summary statistics are stored in AspireObject variables
  2. Summary statistics are computed across the document, even if there are multiple analyzers executed multiple times
- **Parent Document Scope** - Also called "Parent Job Scope", which accumulates statistics across multiple documents.
  1. Summary statistics are stored into the parent job (technically, stored into the parent job's AspireDocument object).
  2. Summary statistics are computed across multiple documents or sub-jobs.

3. For example, if the parent job is a large XML file of multiple documents which are then split with an XML Sub Job Extractor - you can compute statistics over all of the sub jobs by using parent job scope.

- **Grandparent Scope** - Accumulates statistics across all of the documents across all of the parents within the same grandparent job. Note that this requires an Aspire configuration with three levels (i.e. grandparent job, with two sub-job extractors, one to create parents from the grandparent, and another to create documents for each parent).

Specifying the Scope

Scope are now specified as attributes in the token pipeline. Check out the @scope attribute in the configuration below:

```
<component>
  <processors>
    <processor class="com.searchtechnologies.aspire.components.PunctuationTokenizer" />
      <processor class="org.apache.lucene.analysis.LowerCaseFilter" />
      <processor class="com.searchtechnologies.aspire.components.TokenDocsHistogram" scope="parent"/>
      <processor class="com.searchtechnologies.aspire.components.TokensHistogram" scope="document"/>
      <processor class="com.searchtechnologies.aspire.components.CountTokens" scope="parent"/>
      <processor class="com.searchtechnologies.aspire.components.CountTokens" scope="document"/>
    </processors>
</component>
```

With the @scope attribute you can precisely specify over what range of text the statistics will be accumulated. For the above configuration:

- (TokenDocsHistogram) A count of documents for each unique token will be accumulated across all documents in the parent job.
    - This is for situations where a single parent job processes multiple documents, using a sub-job extractor, for example.
- (TokensHistogram) A count of all occurrences for each token will be accumulated across each document.
    - This data will be attached to the document's AspireObject object for each sub-job.
- (CountTokens) Token counts will be accumulated at two different scopes, both across all documents within the parent job, and also for each document processed.

Creating the Aspire Analyzer

For all cases above, you create an Aspire Analyzer using the following code:

```
aspireAnalyzer = tokenizerContent.newAspireAnalyzer(job);
```

where "job" is a reference to the current job being processed.

Change since earlier 0.4-SNAPSHOT versions: Previously, you passed the AspireObject (i.e. the Job's object variable) to newAspireAnalyzer(). *This is no longer the case*. Now, you always pass the job object, in all circumstances.

Configuration

Element	Type	Default	Description
processors	parent tag	none	A parent tag which contains a list of token processors. Note that the first processor listed within <processors> must be a tokenizer, and all of the others are token filters or token statistics aggregators.
processors /processor	tag with attributes	none	Specifies each individual token processor in the tokenization pipeline. Note that the order of the <processor> tags is important - since the tokens will be processed in the order specified. Each token processor will be called on each token in turn. See below for a list of token processors currently defined.
processor /@class	string	none	Many "core" component processors are provided with the Tokenization Manager itself. All of these core processors are specified using the Java class name of the processor specified with the @class attribute. See Token Processors to determine which processors are "core" and what Java class name should be used for each.
processor /@componentRef	string	none	Component processors may also be specified as Aspire components. This is the method used for processors which are created outside the Tokenization Manager (the non-core processors) and is often used if customers require custom token processors for special needs. See below for how these components are configured in Aspire. Once the component is configured, use the @componentRef attribute to specify the Aspire component name (may be relative or absolute) of the processor.
processor /@scope	string	none	Can be "analyzer", "document", "parent", or "grandparent". Specifies the scope for variables which are created by this token processor. See the discussion of "scope" above.
processor /@useCases (note: plural 'cases')	string	none	This is a comma-separated list of use cases for which the tokenization processor will be enabled. Each "use case" is a name which can be determined by the configuration. The only use case name which has significance is "default" which is the use case which is enabled when no use case is specified to the newAspireAnalyzer() function. See more information about use cases below.
processor/*	parent tag	none	Many tokenization processors have additional configuration parameters which can be specified in nested tags within the <processor> tag. See Token Processors for more information about what configuration options are available for each token processor. <p/> Note that configuration can not be specified when referencing a token processor by component name with the @componentRef attribute.
tagsToProcess	parent tag	none	*(Only for use as a pipeline stage)* A parent tag which contains a list of nested <tag> elements, which identifies the list of AspireObject XML tags to be sent through the token processing pipeline when this component is used as a pipeline stage.
tagsToProcess/tag /@name	String	none	Identifies the XML tag from the AspireObject whose text will be sent through the token processors. Multiple tags can be specified. They are processed in order.

Specifying Tokenization Processors

There are many different methods for specifying token processors.

Option 1: Call out the Lucene classes directly

If the Lucene class takes no arguments for its constructor, it can be called out directly. For example:

```
<processor class="org.apache.lucene.analysis.LowerCaseFilter"/>
```

Option 2: Call out a "built-in" Aspire Tokenization Manager Class directly

There are a number of token processors built into the Aspire Tokenization manager, which can be called out by class name. For example:

```
<processor class="com.searchtechnologies.aspire.components.CountTokens" scope="document"/>
```

See Token Processors for more information on these token processors.

Option 3: Reference an Aspire component by component name

Some token processors may have very large databases associated with them. The best example is the extractor, which takes a token stream and looks it up in a dictionary. In order to avoid opening up the dictionary multiple times, it is loaded as an Aspire component in a common area and accessed by reference. For example:

```
<component name="CustomTokenFilter" subType="default" factoryName="aspire-my-custom-processor"/>

.
.
.

<component name="TokenizationMgr" subType="default" factoryName="aspire-tokenizer">
  <processors>
    .
    .
    .
    <processor componentRef="CustomTokenFilter"/>
    .
    .
    .
  </processors>
</component>
```

In the above example, "CustomTokenFilter" component is first configured as a standard Aspire <component>. It is then referenced from within the tokenization manager using the @componentRef attribute

If the componentRef is an absolute path, such as "/common/CustomTokenFilter", this allows you to share Aspire token filter factories or Aspire tokenizer factories across multiple instances of the Tokenization Manager.

Use Cases

Complex text analytics applications will often have many different but mostly similar tokenization pipelines. Further, these pipelines can be very long and contains dozens and dozens of stages.

"Use cases" was invented to handle this situation in a graceful fashion. With use cases, all token processors for multiple similar pipelines can be merged together into a single (comprehensive) pipeline. Small changes to the overall pipeline can be tagged with "use case" names, which allow for individual processors to be included only in certain situations.

As an example, consider the pipeline below:

```
<component>
  <processors>
    <processor class="com.searchtechnologies.aspire.components.PunctuationTokenizer" />
    <processor class="org.apache.lucene.analysis.LowerCaseFilter" useCases="italy,uk"/>
    <processor class="com.searchtechnologies.aspire.components.TokensHistogram" scope="analyzer" useCases="italy,
default"/>
    <processor class="com.searchtechnologies.aspire.components.TokenDocsHistogram"  scope="analyzer" useCases="
uk"/>
    <processor class="com.searchtechnologies.aspire.components.CountTokens" scope="analyzer" useCases="default"/>
  </processors>
</component>
```

The @useCases attribute identify the use cases for which the processor applies. If the @useCases attribute is missing, the token processor will apply to all use cases.

Using the above example, the following code for the "italy" use case:

```
AspireAnalyzer aa = s.newAspireAnalyzer("italy");
```

will generate a pipeline with the following stages:

- PunctuationTokenizer
- LowerCaseFilter
- TokensHistogram

The following code constructs a pipeline for the "uk" use case:

```
AspireAnalyzer aa = s.newAspireAnalyzer("uk");
```

and creates the following pipeline:

- PunctuationTokenizer
- LowerCaseFilter
- TokenDocsHistogram (Note: this module is different from the "italy" use case)

The "default" Use case

The special case "default" will construct a pipeline for the default case. Default use case can be specified when creating a new analyzer in two ways, either the no-argument method:

```
AspireAnalyzer aa = s.newAspireAnalyzer();
```

or by specifying "default" specifically:

```
AspireAnalyzer aa = s.newAspireAnalyzer("default");
```

The "default" editor will automatically include all token processing modules where the @useCases attribute is missing, OR those which are specifically tagged with the use case "default". In the above example, this would include the following modules:

- PunctuationTokenizer
- com.searchtechnologies.aspire.components.TokensHistogram
- com.searchtechnologies.aspire.components.CountTokens

Notice that, in the above example, the CountTokens module is used *only* only for the default case. Because the module has an @useCases attribute, it no longer applies to all use cases automatically.

Example Configuration

Simple

```
<component name="TokenizationMgr" subType="default" factoryName="aspire-tokenizer">
  <processors>
    <processor class="com.searchtechnologies.aspire.components.PunctuationTokenizer"/>
    <processor class="org.apache.lucene.analysis.LowerCaseFilter"/>
    <processor class="com.searchtechnologies.aspire.components.CountTokens" scope="document"/>
    <processor class="com.searchtechnologies.aspire.components.TokenDocsHistogram" scope="parent"/>
  </processors>
</component>
```

Complex

```
<component name="TokenizationManager" subType="default" factoryName="aspire-tokenizer">
  <tagsToProcess>
    <tag name="title"/>
    <tag name="content"/>
  </tagsToProcess>

  <processors>
    <processor class="com.searchtechnologies.aspire.components.PunctuationTokenizer">
      <exceptions>-:'</exceptions>
    </processor>
    <processor class="org.apache.lucene.analysis.LowerCaseFilter"/>
    <processor class="com.searchtechnologies.aspire.components.CountTokens" scope="document"/>
    <processor class="com.searchtechnologies.aspire.components.TokensAndPairs"/>
    <processor class="com.searchtechnologies.aspire.components.TokensHistogram" scope="document"/>
    <processor class="com.searchtechnologies.aspire.components.TokenDocsHistogram" scope="parent"/>
    <processor class="com.searchtechnologies.aspire.components.GatherTokenStatistics" scope="parent"/>
    <processor class="com.searchtechnologies.aspire.components.ScoreDocWithTokenPercentiles" scope="document">
      <tokenStatsFile tokenFieldNum="3" observationsCountFieldNum="2" minObservations="50">
        data/token-pairs-dictionary.txt
      </tokenStatsFile>
    </processor>
  </processors>
</component>
```

Example Use Within A Pipeline

The above configuration for the tokenization manager can be used inside of a pipeline as a simple pipeline stage.

```
<pipeline name="process-patent" default="true">
  <stages>
    <stage component="FetchUrl" />
    <stage component="ExtractText" />
    <stage component="TokenizationManager"/>
  </stages>
</pipeline>
```

This configuration will automatically send all of the text in the XML tags configured in <tagsToProcess> through the configured tokenization pipeline.

Typical Usage in Groovy Scripts

First, assume that you have the following tokenization manager defined:

```
  <component name="TokenizationMgr" subType="default" factoryName="aspire-tokenizer">
    <processors>
      <processor class="com.searchtechnologies.aspire.components.PunctuationTokenizer"/>
      <processor class="org.apache.lucene.analysis.LowerCaseFilter"/>
      <processor class="com.searchtechnologies.aspire.components.CountTokens" scope="document"/>
      <processor class="com.searchtechnologies.aspire.components.TokenDocsHistogram" scope="document"/>
    </processors>
  </component>
```

With the above tokenization manager, you can use the following Groovy Script:

```
<component name="processCVContent" subType="default" factoryName="aspire-groovy">
  <variable name="tokenizerContent" component="TokenizationMgr"/>
  <script>
    <![CDATA[

      // Get an analyzer with scope = Document-level scope
      def aspireAnalyzer = tokenizerContent.newAspireAnalyzer(job);
      aspireAnalyzer.processAll(docText);

      // You can get the counts using aspireAnalyzer.get():
      def count = aspireAnalyzer.get("tokenCount");

      // Note that "tokenCount" above is computed by the
      // "com.searchtechnologies.aspire.components.CountTokens" class specified
      // in the token processing pipeline above

      // Also, you can get the counts just by referencing the variables directly
      // (this works if scope = document scope or parent document scope)
      def count2 = tokenCount;
      println "Count of unique tokens:  " + tokenDocsHistogram.size()

      // "tokenDocsHistogram" is a variable put on the document by the
      // "com.searchtechnologies.aspire.components.TokenDocsHistogram" class specified in
      // the tokenization manager configuration above
    ]]>
  </script>
</component>
```